Sorting

Using ADTs to Implement Sorting Algorithms
Objectives

• Examine several sorting algorithms that can be implemented using collections and in-place:
  
  * **Insertion Sort**
  * **Selection Sort**
  * **Quick Sort**

• Analyze the time complexity of these algorithms
Sorting Problem

• Suppose we have an unordered list of objects that we wish to have sorted into ascending order
• We will discuss the implementation of several sort methods with a header of the form:

```java
public void someSort( UnorderedList list)
// precondition: list holds a sequence of objects in some random order
// postcondition: list contains the same objects, now sorted into ascending order
```
Comparing Sorts

• We will compare the following sorts:
  • *Insertion Sort* using stacks and in-place
  • *Selection Sort* using queues and in-place
  • *Quick Sort*

• Assume that there are $n$ items to be sorted into ascending order
Insertion Sort

- **Insertion Sort** orders a sequence of values by repetitively inserting the next value into a sorted subset of the sequence

- More specifically:
  - Consider the first item to be a sorted subsequence of length 1
  - Insert the second item into the sorted subsequence, now of length 2
  - Repeat the process, always inserting the first item from the unsorted portion into the sorted subsequence, until the entire sequence is in order
Insertion Sort Algorithm

**Example:** sorting a sequence of Integer objects

Value 5 is to be inserted where the 8 is: reference to 8 will be copied to where the 5 is, the 5 will be put in the vacated position, and the sorted subsequence now has length 2.
2 will be inserted here

5 8 2 6 9 4 6

2 5 8 6 9 4 6

6 will be inserted here

2 5 8 6 9 4 6

2 5 6 8 9 4 6
9 will be inserted here

4 will be inserted here
6 will be inserted here

And we’re done!
Insertion Sort using Stacks

Approach to the problem:

• Use two temporary stacks sorted and temp, both of which are originally empty

• The contents of sorted will always be in order, with the smallest item on the top of the stack
  • This will be the “sorted subsequence”

• temp will temporarily hold items that need to be “shifted” out in order to insert the new item in the proper place in sorted
**Algorithm** insertionSort \((A,n)\)

**In:** Array \(A\) storing \(n\) elements

**Out:** Sorted array

sorted = empty stack

temp = empty stack

for \(i = 0\) to \(n-1\) do {
    while (sorted is not empty) and (sorted.peek() < A[i]) do
        temp.push(sorted.pop())
    sorted.push(A[i])
    while temp is not empty do
        sorted.push(temp.pop())
}

for \(i = 0\) to \(n-1\) do
    A[i] = sorted.pop()
Insertion Sort

sorted

temp
Insertion Sort

8 5 2 6 9 4 6

sorted

8

temp
Insertion Sort

8 5 2 6 9 4 6

sorted

5 8
temp
Insertion Sort

sorted  temp

8  5  2  6  9  4  6
Insertion Sort

sorted

2
5
8
temp
Insertion Sort

sorted  |  temp
---|---
8  |  2
5  |  6
2  |  9
6  |  4
8  |  6

5
2
Insertion Sort

sorted 8 5 2 6 9 4 6

temp 8 2 5 6
Insertion Sort

sorted  
\[
\begin{array}{ccccccc}
8 & 5 & 2 & 6 & 9 & 4 & 6 \\
\end{array}
\]

temp  
\[
\begin{array}{cc}
6 & 8 \\
5 & 2 \\
\end{array}
\]
Insertion Sort

sorted

8 5 2 6

temp

8 5 6 2
Insertion Sort

sorted

2
5
6
8

temp

8 5 2 6 9 4 6
Insertion Sort
Insertion Sort

sorted: 8 5 2 6 | temp: 6 8 | 9 4 6
Insertion Sort
Insertion Sort

sorted

temp

8 5 2 6 9 4 6

8 6 5 2
Insertion Sort
Insertion Sort

8 5 2 6 9 4 6

sorted

8
9

temp
6
5
2
Insertion Sort

8 5 2 6 9 4 6

sorted
temp

6 8 9

5 2
Insertion Sort

8 5 2 6 9 4 6

sorted

5 6 8
9
temp

2
Insertion Sort

sorted

2
5
6
8
9

temp

8 5 2 6 9 4 6
Insertion Sort

sorted: 8 5 2 6 9 4 6

temp: 2 4 5 6 6 8 9
Insertion Sort

sorted temp

2 5 2 6 9 4 6

diagram showing the insertion sort process with elements being inserted into the sorted array.
Insertion Sort
Insertion Sort

sorted  temp
Discussion

• Is there a best case?
  • Yes: the items are already sorted, but in reverse order (largest to smallest)
  • What is the time complexity then?

• What is the worst case?
  • The items are already sorted, in the correct order!!
  • Why is this the worst case?
In-Place Insertion Sort

In-Place: the algorithm does not use auxiliary data structures.
In-Place Insertion Sort
In-Place Insertion Sort
In-Place Insertion Sort

sorted

[diagram showing the process of insertion sort with elements 5, 8, and 2 being inserted into the sorted part of the array]
In-Place Insertion Sort
In-Place Insertion Sort

sorted

2 5 6 8 9 4 6

9
In-Place Insertion Sort

2 5 6 8 9 4 6

sorted
In-Place Insertion Sort

sorted

2 4 5 6 8 9 6
In-Place Insertion Sort

2 4 5 6 8 9 6

sorted
In-Place Insertion Sort

sorted

2 4 5 6 6 → 8 → 9
Algorithm insertionSort (A,n)
In: Array A storing n values
Out: {Sort A in increasing order}
for i = 1 to n-1 do {
    // Insert A[i] in the sorted sub-array A[0..i-1]
    temp = A[i]
    j = i - 1
    while (j >= 0) and (A[j] > temp) do {
        j = j - 1
    }
    A[j+1] = temp
}
Selection Sort

• *Selection Sort* orders a sequence of values by repetitively putting a particular value into its *final* position

• More specifically:
  • Find the *smallest value* in the sequence
  • Switch it with the value in the *first position*
  • Find the *next smallest value* in the sequence
  • Switch it with the value in the *second position*
  • Repeat until all values are in their proper places
Selection Sort Algorithm

Initially, the *entire* container is the “*unsorted portion*” of the container.

Sorted portion is coloured *red*.

Find smallest element in unsorted portion of container

Interchange the smallest element with the one at the front of the unsorted portion

Find smallest element in unsorted portion of container
Interchange the smallest element with the one at the front of the unsorted portion

Find smallest element in unsorted portion of container

Interchange the smallest element with the one at the front of the unsorted portion
Find smallest element in unsorted portion of container

Interchange the smallest element with the one at the front of the unsorted portion

After n-1 repetitions of this process, the last item has automatically fallen into place
Selection Sort Using a Queue

**Approach to the problem:**

- Create a queue **sorted**, originally empty, to hold the items that have been sorted **so far**
- The contents of **sorted** will always be in order, with new items added at the end of the queue
Selection Sort Using Queue Algorithm

• While the unordered list list is not empty:
  • remove the smallest item from list and enqueue it to the end of sorted
• The list is now empty, and sorted contains the items in ascending order, from front to rear
• To restore the original list, dequeue the items one at a time from sorted, and add them to the rear of list
Algorithm selectionSort(list)

temp = empty queue
sorted = empty queue

while list is not empty do {
    smallestSoFar = remove first item from list
    while list is not empty do {
        item = remove first item from list
        if item < smallestSoFar {
            temp.enqueue(smallestSoFar)
            smallestSoFar = item
        } else temp.enqueue(item)
    }
    sorted.enqueue(smallestSoFar)
}

while temp is not empty do
    add temp.dequeue() to the end of list
}

while sorted is not empty do
    add sorted.dequeue() to the end of list
Selection Sort is an \(O(n^2)\) algorithm
The analysis is similar to that of Insertion Sort. We will leave it as an exercise for you to analyze this algorithm.
Discussion

• Is there a best case?
  • No, we have to step through the entire remainder of the list looking for the next smallest item, no matter what the ordering

• Is there a worst case?
  • No
In-Place SelectionSort

Selection sort without using any additional data structures. Assume that the values to sort are stored in an array.

[Diagram showing an array of numbers: 8, 5, 2, 6, 9, 4, 6]
In-Place SelectionSort

First find the smallest value
In-Place SelectionSort

Swap it with the element in the first position of the array.

smallest value

swap
In-Place SelectionSort

Swap it with the element in the first position of the array.

2 5 8 6 9 4 6
In-Place SelectionSort

sorted

2 5 8 6 9 4 6
In-Place SelectionSort

Now consider the rest of the array and again find the smallest value.
In-Place SelectionSort

Swap it with the element in the second position of the array, and so on.
In-Place SelectionSort
In-Place SelectionSort

sorted

smallest value
In-Place SelectionSort

sorted

swap

smallest value
In-Place SelectionSort
In-Place Selection Sort

sorted

smallest value
In-Place Selection Sort

sorted

2 4 5 6 6 8 9
Algorithm selectionSort (A,n)  
In: Array A storing n values  
Out: {Sort A in increasing order}  
for i = 0 to n-2 do {  
    // Find the smallest value in unsorted subarray A[i..n-1]  
    smallest = i  
    for j = i + 1 to n - 1 do {  
            smallest = j  
    }  
    // Swap A[smallest] and A[i]  
    temp = A[smallest]  
    A[i] = temp  
}
Quick Sort

• **Quick Sort** orders a sequence of values by *partitioning* the list around one element (called the **pivot** or **partition element**), then sorting each partition

• More specifically:
  • Choose one element in the sequence to be the **pivot**
  • Organize the remaining elements into three groups (**partitions**): those *greater than* the pivot, those *less than* the pivot, and those *equal* to the pivot
  • Then sort each of the first two partitions (recursively)
Quick Sort

**Partition element** or *pivot*:

- The choice of the *pivot* is arbitrary
- For efficiency, it would be nice if the pivot divided the sequence roughly in half
  - However, the algorithm will work in any case
Quick Sort

**Approach to the problem:**

- We put all the items to be sorted into a container (e.g. an array)
- We choose the pivot (partition element) as the first element from the container
- We use a container **smaller** to hold the items that are smaller than the pivot, a container **larger** to hold the items that are larger than the pivot, and a container **equal** to hold the items of the same value as the pivot
- We then **recursively** sort the items in the containers **smaller** and **larger**
- Finally, copy the elements from **smaller** back to the original container, followed by the elements from **equal**, and finally the ones from **larger**
QuickSort
QuickSort

pivot or partition element

smaller

larger

equal
QuickSort

pivot or partition element

smaller

equal

larger
QuickSort

pivot or partition element

smaller

larger

equal
QuickSort

pivot or partition element

smaller

larger

equal

6 3 2 6 9 4 8

3 2

6
QuickSort

pivot or partition element

6 3 2 6 9 4 8

3 2

9

smaller

larger

equal

6 6
QuickSort

pivot or partition element

smaller

larger

equal
QuickSort

pivot or partition element

6 3 2 6 9 4 8

smaller

3 2 4

larger

9 8

equal

6 6
QuickSort

Sort this list

smaller
3
2
4

larger
9
8

equal
6
6
QuickSort

Sort this list

smaller

larger

6 3 2 6 9 4 8

2 3 4 6 6 9 8

Sort this list

equal

6 6
QuickSort

Sort this list
QuickSort

Sort this list
QuickSort

Copy data back to original list
QuickSort

Copy data back to original list

smaller

larger

equal
QuickSort

Copy data back to original list

smaller
2 3 4

larger
8 9

equal
6 6
QuickSort

Copy data back to original list

smaller

larger

2 3 4

8 9

6 6

6 6

2 3 4

8 9
QuickSort

sorted!

smaller

larger

equal

2 3 4
8 9

6 6
8 9
QuickSort

How to sort this list?

smaller

larger

9 8

6 6
QuickSort

3
2
4

pivot

smaller

larger

equal
QuickSort

smaller

2

larger

4

equal

3

3 2 4
QuickSort

\[ \begin{array}{ccc}
3 & 2 & 4 \\
\downarrow & & \\
\text{smaller} & & \\
2 & & \\
\downarrow & & \\
\text{labeled} & & \\
4 & & \\
\downarrow & & \\
\text{larger} & & \\
\end{array} \]

sort lists

\[ \begin{array}{c}
3 \\
\downarrow \\
\text{equal} \\
\end{array} \]
QuickSort

- 2
- 3
- 4

smaller

2

larger

4

copy data back

equal

3
Algorithm quicksort(A,n)

In: Array A storing n values

Out: {Sort A in increasing order}

If n > 1 then {
    smaller, equal, larger = new arrays of size n
    \( n_s = n_e = n_l = 0 \)
    pivot = A[0]
}

\[ \text{pivot} = 3 \]

\[ \text{smaller} \]
\[ \text{equal} \]
\[ \text{larger} \]

\( n_s = 0 \)
\( n_e = 0 \)
\( n_l = 0 \)
**Algorithm** quicksort(A,n)

**In:** Array A storing n values

**Out:** {Sort A in increasing order}

**If** \( n > 1 \) **then** {

smaller, equal, larger = new arrays of size n

\( n_s = n_e = n_l = 0 \)

pivot = A[0]

**for** i = 0 **to** n-1 **do** // Partition the values

}
**Algorithm** quicksort(A,n)

**In:** Array A storing n values

**Out:** {Sort A in increasing order}

**If** n > 1 **then** {

- smaller, equal, larger = new arrays of size n
- \( n_s = n_e = n_l = 0 \)
- pivot = A[0]

**for** i = 0 **to** n-1 **do** // Partition the values

- **if** A[i] = pivot **then** equal[\( n_e++ \)] = A[i]

}
Algorithm quicksort(A,n)
In: Array A storing n values
Out: {Sort A in increasing order}
If \(n > 1\) then {
    smaller, equal, larger = new arrays of size \(n\)
    \(n_s = n_e = n_l = 0\)
    pivot = \(A[0]\)
    for \(i = 0\) to \(n-1\) do // Partition the values
        if \(A[i] = pivot\) then equal[\(n_e++\)] = \(A[i]\)
        else if \(A[i] < pivot\) then smaller[\(n_s++\)] = \(A[i]\)
}
**Algorithm** quicksort(A,n)

**In**: Array A storing n values

**Out**: {Sort A in increasing order}

If n > 1 then {

smaller, equal, larger = new arrays of size n

\(n_s = n_e = n_l = 0\)

pivot = A[0]

for i = 0 to n-1 do  // Partition the values

\[\text{if } A[i] = \text{pivot then equal}[n_e++] = A[i]\]

\[\text{else if } A[i] < \text{pivot then smaller}[n_s++] = A[i]\]

\[\text{else larger}[n_l++] = A[i]\]

}
Algorithm quicksort(A,n)
In: Array A storing n values
Out: {Sort A in increasing order}
If n > 1 then {
    smaller, equal, larger = new arrays of size n
    n_s = n_e = n_l = 0
    pivot = A[0]
    for i = 0 to n-1 do // Partition the values
        if A[i] = pivot then equal[n_e++] = A[i]
        else if A[i] < pivot then smaller[n_s++] = A[i]
        else larger[n_l++] = A[i]
    quicksort(smaller,n_s)
}
**Algorithm** quicksort(A,n)

**In:** Array A storing n values

**Out:** {Sort A in increasing order}

If n > 1 then {

- smaller, equal, larger = new arrays of size n
- \( n_s = n_e = n_l = 0 \)
- pivot = A[0]

for i = 0 to n-1 do  // Partition the values
  - if A[i] = pivot then equal[\( n_e++ \)] = A[i]
  - else if A[i] < pivot then smaller[\( n_s++ \)] = A[i]
  - else larger[\( n_l++ \)] = A[i]

quicksort(smaller,\( n_s \))

quicksort(larger,\( n_l \))

}
Algorithm quicksort(A,n)
In: Array A storing n values
Out: {Sort A in increasing order}
If n > 1 then {
    smaller, equal, larger = new arrays of size n
    ns = ne = nl = 0
    pivot = A[0]
    for i = 0 to n-1 do // Partition the values
        if A[i] = pivot then equal[ne++] = A[i]
        else if A[i] < pivot then smaller[ns++] = A[i]
        else larger[nl++] = A[i]
    quicksort(smaller,ns)
    quicksort(larger,nl)
    i = 0
    for j = 0 to ns do A[i++] = smaller[j]
    for j = 0 to ne do A[i++] = equal[j]
    for j = 0 to nl do A[i++] = larger[j]
}
Analysis of Quick Sort

• We will look at two cases for Quick Sort:
  • **Worst case**
    • When the pivot element is the *largest* or *smallest* item in the container (why is this the worst case?)
  • **Best case**
    • When the pivot element is the *middle* item (why is this the best case?)
Worst Case Analysis:

- We will count the number of operations needed to sort an initial container of \( n \) items.
- Assume that the pivot is the largest item in the container and all values in the array are different.
- \( n \leq 1 \); the algorithm performs just one operation to test that \( n \leq 1 \), so \( O(1) \)
- \( n > 1 \); the pivot is chosen from the container and then the \( n \) items are redistributed into three containers:
  - smaller is of size \( n-1 \)
  - bigger is of size 0
  - equal is of size 1

Moving each item requires a constant number \( c' \) of operations, so this step performs \( n \) operations, so \( O(n) \)
• Then we have two recursive calls:
  • Sort **smaller**, which is of size **n-1**
  • Sort **bigger**, which is of size **0**
• Assume again the pivot is the largest remaining item.
  • The smaller array will have **n-2** (from the original **n**) elements.
• Assume this continues each time the method is called recursively.
• This will run essentially **n**, **O(n)** times.
• So the **worst case** time complexity of Quick Sort is **O(n^2)**
Best Case Analysis

• The best case occurs when the pivot element is chosen so that the two new containers are as close as possible to having the same size

• It is beyond the scope of this course to do the analysis, but it turns out that the best case time complexity for Quick Sort is $O(n \log_2 n)$

• And it turns out that the average time complexity for Quick Sort is the same